Minimum Viable Product

This document describes how to set up and test-drive the Spack / SpackDev MVP1a ("LArSoft edition").

Overview of the MVP

The MVP is intended to allow experienced experimenters to get a first look at the intended build, development and deployment system that is intended to replace UPS and cetbuildtools / MRB. This system is based on Spack, an HPC-originated package management and build orchestration system, and includes a FNAL-authored companion system called SpackDev to facilitate the simultaneous development of multiple packages.

In order to use the MVP, you should choose at least one package that you wish to "develop" *i.e* manipulate the code, build, test and debug. Anything you choose should have as few "external" dependencies as possible beyond the standard LArSoft distribution. Each such dependency that cannot be expected to exist already on the system must have a Spack recipe: we would be happy to work with you to ensure that this is the case. If you wish to get a feel for the system without doing too much work, you may choose any or all of the art suite or LArSoft packages to develop.

To get help at any point or provide feedback on this documentation or the MVP itself, please send email to spackdev-team. Subjects could include (but are not limited to): * Clarifying / improving the documentation. * Production of recipes or DAG extensions for your package(s) and their dependencies (see below). * Converting your package from cetbuildtools to cetmodules. * Workflow / usability issues. * Design / conceptual questions or problems.

Limitations and Requirements

- The MVP is currently limited to one platform (SL7 and related), one compiler (GCC 7.3.0) and one C++ standard (C++17).
- Only the BASH shell is supported currently; other advanced Bourne-type shells (such as zsh) *may* work, but are not guaranteed.
- All packages (including the compiler) must be built in-place.
- The products available as dependencies are currently somewhat limited from the point of view of, say, LArSoft, but products can be mde available as necessary.
- Your system must pass the checkPrerequisites test (this will be enforced by the bootstrap script). If your system is missing prerequisites, please have your system administrator install the missing RPMs.
- At least 12GiB of free space is required, and several hours are required for dependencies to be built.
- Packages to be developed with SpackDev must (at this time) be buildable with CMake. If they are built using `cetbuildtools`, then they

must be retrofitted to use `cetmodules` instead (see below for details), as `cetbuildtools` implicitly depends upon UPS for its operation.

Quick start

The following will get you going with a package from LArSoft. Note that the `spack dev init` command will take a significant amount of time (an hour or so on a reasonably fast machine).

- Obtain the bootstrap script.
- `chmod +x bootstrap-mvp`
- `./bootstrap-mvp -j <ncores> -v <scratch-dir>/MVP`
  (`<scratch-dir>` must exist; `MVP` must either not exist or be empty).
- `cd MVP`
- `. setup.sh`
- `spack dev init -b spackdev-larsoft --default-branch=MVP1a \`
  `--dag-file ../spack_glue/MVP/templates/larsoft-dag.txt \`
  `-v larsim`
- `. spackdev-larsoft/spackdev-aux/env/env.sh`
- `cd spackdev-larsoft/build`
- `CTEST_PARALLEL_LEVEL=<ncores> cmake --build . -j <ncores>`
- `spackdev env --cd --prompt larsim`
- `ctest -j<#>`

Bootstrapping a new area

- Download the bootstrap-mvp script (*e.g.*):

  ```
  curl -O -J -L --fail https://cdcvs.fnal.gov/redmine/projects/\
  spack-planning/repository/revisions/master/raw/MVP/bootstrap-mvp
  chmod +x bootstrap-mvp
  ```

- Execute `bootstrap-mvp -h` to understand available options. In short:
  - The non-option argument should be the top directory of the MVP area. If not specified, we default to the current working directory. The directory specified should be empty. If it does not exist, it will be created provided its parent exists.
  - If you wish, you may specify out-of-tree directories for the "spack-tools" area containing the compiler, and for the "spack-data" area which will contain the bulk of the packages you build. You would do this if you wanted to put reproducible "scratch" data on a different (possibly not backed-up) filesystem than the rest of the MVP area.
  - The first time you run, you may wish to add a `-v` option to see more detailed progress information.

The bootstrap script will create the various areas where you specify, including:

- `spack-tools`, an area containing the compiler and one or two other tools, such as git.
- `spack/`, containing the Spack application, its accompanying code and all its package recipes
- `fnal-art/`, containing recipes for the art suite, and into which you would place your own recipes.
- `spack-data`, an area containing all the other products to be built.
- `spackdev/`, containing the SpackDev application and its accompanying code.
- `spack_glue/MVP/`, containing the bootstrap script, these instructions and more.
- `setup.sh`, a BASH setup script which, when sourced, will get your environment ready to initialize a SpackDev area and start developing.

The script will take relatively little time to complete (depending on the capabilities of your system) as it should be able to download relocatable binary packages of the tools (such as GCC and its dependencies).

Setting up to use SpackDev to develop your packages

Now all the low level tools are installed, you will need to tell SpackDev about all the packages and dependencies you wish to develop. This will involve a `package.py` file for each package you wish to develop, and a "DAG" file containing the list of all the packages you *might* wish to develop and their dependencies, including all variant and version configuration information. This, while not being a DAG in its own right, will allow Spack to produce a fully-specified DAG with which to ensure that everything will be built as required.

Creating your `package.py` files

Full documentation on the Spack system may be found at https://spack.readthedocs.io/en/latest/index.html, including a getting started guide, packaging guide and command reference.

- In the `spack_glue/MVP/templates` directory is a file `cet_package.py`.
- For each package `XXXX`, make a directory `spack/xxxx` and copy it to `spack/xxxx/package.py`.
- Load `package.py` into an editor. Look for all instances of "`###`" (with a trailing space to avoid comment rule lines) as these denote areas requiring attention. Currently they are:
  - Class name (camel case by convention) and description.
  - Home page URL (normally your Fermi Redmine top-level wiki URL).
  - Git base URL (normally the same as your Fermi Redmine project page URL).
  - The version declaration(s), including name (*e.g.* 'develop'), and branch.

- The `cxxstd` variant declaration, which is only necessary if your package contains C++ code.
- "Build-only" dependencies: packages needed at build time, but not at link or run time.
- "Other" dependencies. Note that these are *direct* dependencies *i.e.* packages whose headers and / or libraries you need in your current package. Indirect dependencies should be handled by those dependencies which need them directly. Specify version or variant requirements for your dependencies here also (see Spack documentation, referenced above).
- The `setup_environment()` and `setup_dependendent_environment()` functions ensure the expected environment is set up for your package, both when it is set up as a top level product (`setup_environment()`) and when it is a a dependency of another product being set up (`setup_dependent_environment()`). Exactly which environment variables need to be propagated in this way will depend on the details of your package.

Specifying the full dependency tree for all packages and dependencies

An example file (`spack_glue/MVP/templates/larsoftdag.txt` is shown below, with explanation following:

```
larsoft@MVP1a \
  ^wirecell@0.10.9 cxxstd=17 ^jsoncpp@1.7.7 cxxstd=17 \
  ^jsonnet@0.11.2 cxxstd=17 ^dk2nudata@01_07_02 cxxstd=17 \
  ^dk2nugenie@01_07_02 cxxstd=17 ^genie@2_12_10 cxxstd=17 \
  ^lhapdf@5.9.1 cxxstd=17 ^pdfsets@2.8.6 ^log4cpp@1.1.3 cxxstd=17 \
  ^cry@1.7 cxxstd=17 ^geant4@10.03.p03~data cxxstd=17 \
  ^xerces-c@3.2.2 cxxstd=17 ^ifdh-art@MVP1a cxxstd=17 \
  ^ifdhc@2.3.10 cxxstd=17 ^ifbeam@2.3.0 cxxstd=17 ^libwda@2.26.0 \
  ^nucondb@2.3.0 ^art-root-io@MVP1a cxxstd=17 ^art@MVP1a cxxstd=17 \
  ^gallery@MVP1a cxxstd=17 ^canvas-root-io@MVP1a cxxstd=17 \
  ^root@6.16.00~root7+davix+fits+fortran+mysql+sqlite+postgres+python+ssl \
  +xrootd+fftw+pythia6+tmva cxxstd=17 ^gsl@2.5 ^fftw@3.3.8~mpi~openmp \
  ^intel-tbb@2019.3 cxxstd=17 \
  ^postgresql@9.6.11+client_only+python+perl+threadsafe+gssapi lineedit=libedit \
  ^mysql@5.5.62+client_only cxxstd=17 ^xrootd@4.8.5+python~readline cxxstd=17 \
  ^canvas@MVP1a cxxstd=17 ^py-numpy@1.15.4 ^netlib-lapack@3.8.0 \
  ^pythia6@6.4.28 ^cppunit@1.14.0 cxxstd=17 ^range-v3@0.4.0 cxxstd=17 \
  ^clhep@2.4.1.0 cxxstd=17 ^messagefacility@MVP1a cxxstd=17 \
  ^hep-concurrency@MVP1a cxxstd=17 ^fhicl-cpp@MVP1a cxxstd=17 \
  ^py-pybind11@2.2.4 ^cetlib@MVP1a cxxstd=17 ^cetlib-except@MVP1a cxxstd=17 \
  ^boost@1.69.0 cxxstd=17 ^cetmodules@1.02.04 ^cmake@3.13.2 \
  ^libxml2@2.9.9+python ^python@2.7.15 ^sqlite@3.26.0 ^catch~single_header
critic@MVP1a
```

```
ninja@1.8.2
cppgsl@2.0.0 cxxstd=17
```

This file is in the (somewhat arcane) format expected by a `spack install` command. Basically, it is:

```
<package> <variants>... ^<dependency> <dep-variants> ^...
...
```

Subsequent lines indicate packages that are not in the dependency trees of other package specifications, but which may share depdenencies in common. These common dependencies do not have to be specified a second time–they will be harmonized. Be sure to specify any non-default variants on the command line, not only for the top level package but for any dependencies.

Developing a single package If you have one package you wish to develop, which (say) depends on `larsoft`, you would specify it ahead of `larsoft` on the same line with any variants, and `^`-labeled direct and indirect dependencies, and prefix the reference to `larsoft` with `^` to identify it as a dependency. If your package does not depend directly on `larsoft`, but they have dependencies in common, then your package and any dependencies *not* in common with larsoft would be specified on a separate line.

Developing multiple packages If one of the packages you wish to develop is a "common dependent" which can count every other package as a (direct or indirect) dependency, then all your other packages should be specified as dependencies of the top-level package (with a leading `^`). Otherwise, use as many separate entries as you need, following the prescription for the single package case, above.

Initializing a SpackDev area

Once you have recipes and a fully-specified DAG file, you are ready to initialize a SpackDev development area.

Spack (and git, and others) is an application that has subcommands. If you have bootstrapped the MVP successfully, one of those subcommands is `dev`. It also has subcommands, which may be listed by invoking the usage help with `spack dev -h`. The first one you will need is `spack dev init`. You may get more information about how to invocate any subcommand with `spack dev <subcommand> -h`. Simply stated however, the usage is:

```
spack dev init -b <base-dir> --dag-file <dag-file> \
--default-tag=MVP1a <package>+
```

where:

- `<base-dir>` is the directory you wish to serve as the root of your development area (current working directory if not specified).

- `<dag-file>` is the file you generated earlier.
- `<package>+` are all the packages you wish actually to develop together, in the form `<package>@<tag>` or `<package>^<branch>`.
- `--default-tag` specifies the tag to be used for packages for which you have not explicitly specified a branch or tag. See the usage help for more details.

SpackDev is capable of using the DAG created from your dag-file to identify any packages you may have missed inbetween the ones you mentioned which must be checked out and built locally in order to provide a consistent whole. Note however that recipes must exist for all of these, and any other "external" dependencies outside those you wish to develop in a SpackDev area, and that they must be properly included in the DAG file.

Note: one consequence of the current state of the project is that the initialization process is somewhat time-consuming, even discounting the time taken to build all the dependencies.

When your SpackDev development area has been initialized, you will see inside it:

- `build/`: where the build takes place.
- `bin/`: links to tools such as `cmake` or `ninja`.
- `install/`, where products are installed.
- `srcs/`: all checked-out sources appear here. This also contains the generated `CMakeLists.txt` file.
- `tmp/`: CMake-generated temporary files, such as target timestamps.
- `spackdev-aux/`: containing tool wrappers and environment information for each package.

Developing in your SpackDev area.

A successful execution of `spack dev init` will culminate in the invocation of `cmake -G<generator> ../srcs` in the build directory, thereby generating a `Makefile`. After following the instruction to source the `env.sh` file to initialize your environment, you should be able to invoke:

```
CTEST_PARALLEL_LEVEL=<ncores> cmake --build <build-dir> -j <N>
```

where `<build-dir>` is the top-level build directory mentioned above. `CTEST_PARALLEL_LEVEL=<ncores> make -jN` or `ninja` as appropriate from inside `<build-dir>` will also have the desired effect. Within the global build, each package being developed is built, tested and installed separately, and dependencies are satisfied from the installation rather than the build areas. `make help` in the top level build directory will give one a list of targets, such as `<package>` or `<package>-configure`.

Most of your time however, will be spent developing one package at a time, writing code, compiling, testing and debugging. Another SpackDev command

now becomes useful:

```
spack dev build-env [--cd] [--prompt] <package> [<cmd>...]
```

This will start a subshell (or run a command) with the correct environment for the package you wish to develop. Optionally you can be placed in the appropriate build subdirectory (`--cd`) or have a colored label added to your command prompt indicating the current package being developed (`--prompt`). Notes:

- The build area for the current passage should have at least a top level `Makefile` (or `ninja.build`) Executing `make <package>-configure` at the global level should generate this, assuming your `CMakeLists.txt` files are correct.
- You should always exit this environment before executing a global build to avoid possible inconsistencies.

Addendum: converting `cetbuildtools` packages to use `cetmodules`

Below we show for illustration the changes made to certain `CMakeLists.txt` files in the `art` package to transition from `cetbuildtools` to `cetmodules`. In broad brush, the changes are:

- To the top-level `CMakeLists.txt`:
  - Changes to the `cmake_minimum_required()` directive. A suitable invocation might be:
    `cmake_minimum_required(VERSION 3.12...3.14 FATAL_ERROR)`
    The minimum version should be at least 3.12; the upper end of the range represents the highest CMake version for which all policies should be considered `NEW` unless otherwise specified.
  - Version information is now specified in the `project()` call in standard dot notation rather than in the `product_deps` file in UPS notation. Note that the `product_deps` file is now vestigial and should be removed from the `ups/` directory. If you have any special notations in the CMake template files in this directory contact us for help, otherwise these files should be removed also.
  - The `cetmodules` code is located with a simple `find_package(cetmodules REQUIRED)` directive.
  - `find_ups_XXX()` calls are obsoleted in favor of standard CMake `find_package()` calls; minimum version requirements should be specified in dot notation. Some packages such as `CLHEP` or `ROOT` understand specification of individual components to reduce clutter. For those packages for which `find_package()` does not work, `cetmodules` provides the `cet_find_library()` and `cet_find_simple_package()` funcions.
  - Additions to `CMAKE_MODULE_PATH` are necessary only for modules your own package provides for its own purposes: others will be located via

find_package()
  - The UseCPack facility is no longer required.
  - Specification of non-default or not-usually-required installation directories should be set with appropriate CMake set() commands such as:
    set(fcl_dir job)
    set(gdml_dir gdml)
    set(fw_dir fw)

- To ups/CMakeLists.txt:
  - The only directive required here is cet_cmake_config(). Its presence here is historical, and could perfectly well be moved up to the top-level CMakeLists.txt file, provided it comes last, including after any and all subdirectory excursions.
- To CMakeLists.txt files where code is compiled and linked:
  - Generally speaking, references to other cetmodules libraries on the link line should be in "target-style" rather than "variable-style" notation.
  - The format of the variables representing the ROOT component libraries has changed (*e.g.* ROOT_HIST to ROOT_Hist_LIBRARY) . One is encouraged instead to use target notation (*e.g.*, ROOT::Core, ROOT::Physics, *etc.*).
  - Similarly, links requiring Boost and the Intel Threading Building Blocks (TBB) library should now be referred to in target notation as dictated by their respective native CMake configuration files as loaded by find_package().

For example, the differences between the CMakeLists.txt files of the art package between cetbuildtools and cetmodules are:

```
diff --git a/CMakeLists.txt b/CMakeLists.txt
index 09526f0..b6cc1c5 100644
--- a/CMakeLists.txt
+++ b/CMakeLists.txt
@@ -12,18 +12,18 @@
 #  make package (builds distribution tarfile)
 # ======================================================================

+# Required to keep CMake happy.
+cmake_minimum_required(VERSION ${CMAKE_VERSION} FATAL_ERROR)
+# Actual version requirement.
+cmake_policy(VERSION 3.4)

-# use cmake 2.8 or later
-cmake_minimum_required (VERSION 2.8)
+# Project information.
+project(art VERSION 2.11.02 LANGUAGES C CXX)
```

```
-project(art)
+# cetmodules contains our cmake modules
+find_package(cetmodules REQUIRED)

-# cetbuildtools contains our cmake modules
-find_package(cetbuildtools REQUIRED)
-
-list(APPEND CMAKE_MODULE_PATH
-  $ENV{CANVAS_ROOT_IO_DIR}/Modules
-  ${CMAKE_CURRENT_SOURCE_DIR}/Modules)
+list(APPEND CMAKE_MODULE_PATH ${CMAKE_CURRENT_SOURCE_DIR}/Modules)

 include(CetCMakeEnv)
 cet_cmake_env()
@@ -34,30 +34,21 @@ cet_set_compiler_flags(DIAGS VIGILANT
   EXTRA_FLAGS -pedantic
   EXTRA_CXX_FLAGS -Wnon-virtual-dtor -Wdelete-non-virtual-dtor)

-cet_have_qual(sse2 SSE2)
-if (SSE2)
-  cet_add_compiler_flags(CXX -msse2 -ftree-vectorizer-verbose=2)
-endif()
-
 cet_report_compiler_flags()

-# these are minimum required versions, not the actual product versions
-find_ups_product(canvas_root_io)
-find_ups_product(canvas v1_06_00)
-find_ups_product(messagefacility)
-find_ups_product(fhiclcpp)
-find_ups_product(cetlib v2_01_00)
-find_ups_product(cetlib_except v1_01_00)
-find_ups_product(clhep)
-find_ups_product(cppunit)
-find_ups_product(cetbuildtools v6_00_00) # LIBRARY_OUTPUT_DIRECTORY, etc.
-find_ups_product(sqlite)
-find_ups_product(range)
-find_ups_boost()
-find_ups_root()
-
-# SQLite
-cet_find_library(SQLITE3 NAMES sqlite3_ups PATHS ENV SQLITE_LIB NO_DEFAULT_PATH)
+find_package(canvas_root_io REQUIRED)
+find_package(canvas REQUIRED)
+find_package(cetlib_except REQUIRED)
```

```
+find_package(fhiclcpp REQUIRED)
+find_package(messagefacility REQUIRED)
+find_package(TBB REQUIRED)
+find_package(CLHEP COMPONENTS Matrix Vector Random REQUIRED)
+include_directories(${CLHEP_INCLUDE_DIRS})
+find_package(ROOT REQUIRED COMPONENTS)
+find_package(Boost COMPONENTS date_time filesystem program_options
+             regex system unit_test_framework REQUIRED)
+
+cet_find_library(SQLITE3 NAMES sqlite3 REQUIRED)

 # macros for art_dictionary and simple_plugin
 include(ArtDictionary)
@@ -76,11 +66,9 @@ add_subdirectory(tools)
 # source
 add_subdirectory(art)

-# ups - table and config files
-add_subdirectory(ups)
-
 # CMake modules
 add_subdirectory(Modules)

-# packaging utility
-include(UseCPack)
+# ups - table and config files
+add_subdirectory(ups)
+
diff --git a/ups/CMakeLists.txt b/ups/CMakeLists.txt
index fe33cc7..937d12d 100644
--- a/ups/CMakeLists.txt
+++ b/ups/CMakeLists.txt
@@ -1,6 +1,2 @@

-# create package configuration and version files
-
-process_ups_files()
-
+# Create package configuration and version files.
 cet_cmake_config()
diff --git a/art/Framework/Art/CMakeLists.txt b/art/Framework/Art/CMakeLists.txt
index 8c9a2c2..c16fccd 100644
--- a/art/Framework/Art/CMakeLists.txt
+++ b/art/Framework/Art/CMakeLists.txt
@@ -1,3 +1,5 @@
```

```
+find_package(ROOT REQUIRED COMPONENTS Hist Tree)
+
 # Configure file to handle differences for Mac.
 configure_file(${CMAKE_CURRENT_SOURCE_DIR}/check_libs.cc.in
   ${CMAKE_CURRENT_BINARY_DIR}/check_libs.cc @ONLY
@@ -64,8 +66,8 @@ art_make_library(LIBRARY_NAME art_Framework_Art
   # The following are used for InitRootHandlers
   art_Framework_IO_Root
   art_Framework_IO_Root_RootDB
-  ${ROOT_HIST}
-  ${ROOT_TREE}
+  ${ROOT_Hist_LIBRARY}
+  ${ROOT_Tree_LIBRARY}
   )

 # Build an art exec.
diff --git a/art/Framework/Core/CMakeLists.txt b/art/Framework/Core/CMakeLists.txt
index ffc68fc..4a4fe54 100644
--- a/art/Framework/Core/CMakeLists.txt
+++ b/art/Framework/Core/CMakeLists.txt
@@ -14,8 +14,7 @@ art_make(
   fhiclcpp
   cetlib
   canvas
-  ${CLHEP}
-  ${TBB}
+  TBB::tbb
   )

 install_headers(SUBDIRS detail)
```